Partial Differential Equations in Modelica

3rd September 2013

Contents

Chapter 1

Modelica extension for PDE

Space & coordinates

What should be specified

- Dimension of the problem (1,2 or 3D)
- ?? Coordinates (cartesian, cylindrical, spherical ...) – where this information will be used (if at all):
  - in differential operators as grad, div, rot etc.
  - in visualization of results
  - ?? in computation – perhaps equations should be transformed and the calculation would be performed in cartesian coordinates
- Names of independent (coordinate) variables (x, y, z, r, φ, θ...)

Perhaps all these should be specified within the domain deffinition.
Dimension can be infered from number of return values of shape-function or different properities of the domain in other cases.
The base coordinates would be cartesian and they would be always implicitly defined in any domain. Besides that other coordinate systems could be defined also.
Names of independent variables in cartesian coordinates should be fixed *x, (x,y), (x,y,z)* in 1D, 2D and 3D domains respectively.

Domain & boundary

What should be specified

- the domain where we perform the computation and where equations hold
- boundary and its subsets where particular boundary conditions hold

- normal vector of the boundary

Possible approaches

Parametrization of the domain with shape function and ranges – from The Book (Principles of ...), section 8.5.2

Example from the book:

```
model HeatCircular2D
        import DifferentialOperators2D.*;
        parameter DomainCircular2DGrid omega;
        FieldCircular2DGrid u(domain=omega, FieldValueType=SI.Temperature);
equation
        der (u) = pder (u,D.x2)+ pder (u,D . y 2 )       in omega.interior;
        nder(u) = 0                 in omega.boundary;
end HeatCircular2D;

record DomainCircular2DGrid "Domain being a circular region"
        parameter Real radius = 1;
        parameter Integer nx = 100;
        parameter Integer ny = 100;
        replaceable function shapeFunc = circular2Dfunc "2D circular region";
        DomainGe2D interior(shape=shapeFunc,range={{O,radius},{O,1}},geom= ... )
        DomainGe2D boundary (shape=shapeFunc, range={{radius, radius), { 0,1}} ,
        function shapeFunc = circular2Dfunc "Function spanning circular region";
end DomainCircular2DGrid;

function circular2Dfunc "Spanned circular region for v in range 0..1"
        input Real r,v;
        output Real x,y;
algorithm
        x : = r*cos (2*PI*v);
        y := r*sin(2*PI*v);
end circular2Dfunc;

record FieldCircular2DGrid
        parameter DomainCircular2DGrid domain;
        replaceable type FieldValueType = Real;
        replaceable type FieldType = Real[domain.nx,domain.ny,domain.nz];
        parameter FieldType start = zeros(domain.nx,domain.ny,domain.nz.);
        FieldType Val;
end FieldCircularZDGrid;
```

And modified version, where all numerical stuff (grid, number of points – this should be configured using simulation setup or annotations) omitted, modified `pder` operator, `Field` as Modelica build-in type:

```
model HeatCircular2D
        parameter DomainCircular2D omega(radius=2);
        field Real u(domain=omega, start = 0, FieldValueType=SI.Temperature);
equation
        pder(u,time) = pder(u,x)+ pder(u,y) in omega.interior;
        pder(u,omega.boundary.n) = 0    in omega.boundary;
end HeatCircular2D;

record DomainCircular2D
        parameter Real radius = 1;
    parameter Real cx = 0;
        parameter Real cy = 0;
        function shapeFunc
                input Real r,v;
                output Real x,y;
        algorithm
                x := cx + radius*r * cos(2 * C.pi * v);
                y := cy + radius*r * sin(2 * C.pi * v);
        end shapeFunc;
        Region2D interior(shape = shapeFunc, range = {{O,1},{O,1}});
        Region1D boundary(shape = shapeFunc, range = {{1,1},{0,1}});
end DomainCircular2D;
```

Description by the boundary Domain is defined by closed boundary curve, which may by composed of several connected curves. Needs new operator *interior* and type *Domain2d* (and *Domain1D* and *Domain3d*). (similarly used in FlexPDE – http://www.pdesolutions.com/.)

```
package BoundaryRepresentation
  partial function cur
    input Real u;
    output Real x;
    output Real y;
  end cur;
  function arc
    extends cur;
    parameter Real r;
    parameter Real cx;
    parameter Real cy;
  algorithm
    x:=cx + r * cos(u);
  y:=cy + r * sin(u);
end arc;
function line
  extends cur;
  parameter Real x1;
  parameter Real y1;
  parameter Real x2;
  parameter Real y2;
algorithm
  x:=x1 + (x2 - x1) * u;
  y:=y1 + (y2 - y1) * u;
end line;
function bezier3
  extends cur;
  //start-point
  parameter Real x1;
  parameter Real y1;
  //end-point
  parameter Real x2;
  parameter Real y2;
  //start-control-point
  parameter Real cx1;
  parameter Real cy1;
  //end-control-point
  parameter Real cx2;
  parameter Real cy2;
algorithm
  x:=(1 - u) ^ 3 * x1 + 3 * (1 - u) ^ 2 * u * cx1 + 3 *
      (1 - u) * u ^ 2 * cx2 + u ^ 3 * x2;
  y:=(1 - u) ^ 3 * y1 + 3 * (1 - u) ^ 2 * u * cy1 + 3 *
      (1 - u) * u ^ 2 * cy2 + u ^ 3 * y2;
end bezier3;
record Curve
  function curveFun = line;
  // to be replaced with another fun
  parameter Real uStart;
  parameter Real uEnd;
end Curve;
record Boundary
  constant Integer NCurves;
  Curve curves[NCurves];
  //     for i in 1:(NCurves-1) loop
  //assert(Curve[i].curveFun(Curve[i].uEnd) = Curve[i
     +1].curveFun(Curve[i+1].uStart), String(i)+"th
     curve and "+String(i+1)+"th curve are not
     connected.",level = AssertionLevel.error);
```

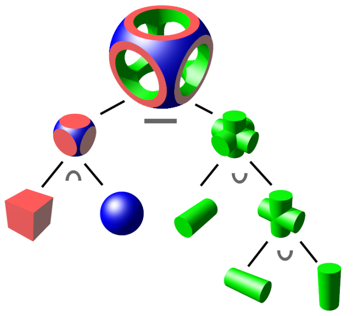

Figure 1.1: constructive solid geometry

```
    //     end for;
    //     assert(curves[NCurves].curveFun(curves[NCurves
        ].uEnd) =
    //                                   curves[1].curveFun(curves
        [1].uStart),
    //                                   String(NCurves)+"th curve
         and first curve are not connected.",
    //                                   level = AssertionLevel.
        error);
  end Boundary;
  record DomainHalfCircle
    constant Real pi = Modelica.Constants.pi;
    arc myArcFun(cx = 0, cy = 0, r = 1);
    Curve myArc(curveFun = myArcFun, uStart = pi / 2,
        uEnd = (pi * 3) / 2);
    line myLineFun(x1 = 0, y1 = -1, x2 = 0, y2 = 1);
    Curve myLine(curveFun = myLineFun, uStart = 0, uEnd =
         1);
    line myLine2(curveFun = line(x1 = 0, y1 = -1, x2 = 0,
         y2 = 1), uStart = pi / 2, uEnd = (pi * 3) / 2);
    Boundary b(NCurves = 2, curves = {myArc,myLine});
    //new externaly defined type Domain2D and operator
        interior:
    Domain2D d = interior Boundary;
  end DomainHalfCircle;
end BoundaryRepresentation;
```

Constructive solid geometry used in Matlab PDE toolbox, http://en.wikipedia.org/wiki/Constructive_sol

Domain is build from primitives (cuboids, cylinders, spheres, cones, user defined shapes ...) applying boolean operations *union*, *intersection* and *difference.*

How to describe boundaries?

Listing of points – export from CAD

Inequalities

Boundary representation (BRep) (NETGEN, STEP)

Fields

Partial derivative

$\frac{\partial^2 u}{\partial x \partial y}$ ... `pder(u,x,y)`
directional derivative ... `pder(u,omega.boundary.n)`

Equations, boundary and initial conditions

Use the *in* operator to express where equations hold.

1.1 List of new language elements and concepts

domain records `DomainLineSegment1D`, `DomainRectangle2D`, `DomainCircular2D`, `DomainBlock3D` ...

ranges to define parameter ranges for a shape-function (e.g. `range={{0,1},0}`)

region types `Region0D`, `Region1D`, `Region2D`, `Region3D` used in domain records

normal vector to N-1 dimensional region in N dimensional domain. (e.g. `room.left.n`)

field type

`pder()` **operator** for partial and directional derivative

`in` **operator** to define where PDE, boundary conditions and other equations hold

+ **operator** region addition

`grad, div, rot` differential operators

operations on fields all standard operations and functions, also on vector fields

Chapter 2

Numerics

Goals

1. advection equation in 1D and eulerian coordinate, dirichlet BC, explicit solver
2. numann BC
3. automatic dt
4. diffusion or mixed equation
5. implicit solver
6. systems of equations
7. 2D (rectangle), 3D (cube)
8. lagrangian coordinate
9. general domain

difference schemes separated from the rest of solver

Difussion eq:

$$u_t = \alpha u_{xx}$$

or

$$\begin{aligned} u_t &= -w_x \\ w &= -\alpha u_x. \end{aligned}$$

String eq:

$$y_{tt} = k y_{xx}$$

or

$$\begin{aligned} s_x &= kv_t \\ y_t &= v \\ y_x &= s \end{aligned}$$

The description without higher derivative is ugly, we need higher derivatives.

Representation

Explicit

$$u_t = f(u, u_x, t) \tag{2.1}$$

resp.

$$u_t = f(u, u_x, u_{xx}, \dots, t)$$

..

Implicit

$$F(u, u_t, u_x, t) = 0 \tag{2.2}$$

resp.

$$F(u, u_t, u_x, u_{xx}, \dots, t) = 0$$

Solvers

Difference schemes for explicit solver

U denotes discretized u

Time difference from Lax-Friedrichs in explicit form (i.e. with the u_m^{n+1} on LHS):

$$u_m^{n+1} = D_t^{exp}(v, U, n, m) = v\Delta t + \frac{1}{2}(u_{m+1}^n + u_{m-1}^n) \tag{2.3}$$

Space difference from Lax-Friedrichs:

$$D_x(U, n, m) = \frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x} \tag{2.4}$$

Explicit solver Lax-Friedrichs

We solve equation (2.1) substituing space difference (2.4) and applying time difference in explicit form (2.3):

$$\begin{aligned} u_m^{n+1} &= D_t^{exp}(f((u_m^n, D_x(U, n, m), t^n))) = \\ &= \Delta t \cdot f(u, \frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x}, t) + \frac{1}{2}(u_{m+1}^n + u_{m-1}^n) \end{aligned}$$

Difference schemes for implicit solver space difference from Crank-Nicolson

$$
\begin{aligned}
D_x(u^n_{m-1}, u^n_m, u^n_{m+1}, u^{n+1}_{m-1}, u^{n+1}_m, u^{n+1}_{m+1}) &= \frac{1}{2}\left(\frac{u^{n+1}_{m+1} - u^{n+1}_{m-1}}{2\Delta x} + \frac{u^n_{m+1} - u^n_{m-1}}{2\Delta x}\right)\\
D_{xx}(u^n_{m-1}, u^n_m, u^n_{m+1}, u^{n+1}_{m-1}, u^{n+1}_m, u^{n+1}_{m+1}) &= \frac{1}{2}\left(\frac{u^{n+1}_{m+1} - 2u^{n+1}_m + u^{n+1}_{m-1}}{2(\Delta x)^2} + \frac{u^n_{m+1} - 2u^n_m + u^n_{m-1}}{2(\Delta x)^2}\right)
\end{aligned}
\tag{2.5}
$$

time difference from Crank-Nicolson

$$
D_t(u^n_{m-1}, u^n_m, u^n_{m+1}, u^{n+1}_{m-1}, u^{n+1}_m, u^{n+1}_{m+1}) = \frac{u^{n+1}_m - u^n_m}{\Delta t} \tag{2.6}
$$

Implicit solver Crank-Nicolson
With nonlinear solver:
We solve equation (2.2) substituting space (2.5) and time (2.6) differences

$$
F\left(u^n_m, D_t(u^n_{m-1}, ...), D_x(u^n_{m-1}, ...), t^n\right) = 0,\ m \in \hat{M} \tag{2.7}
$$

and than solving the whole system for all unknown $u^{n+1}_{:}$. System has 3-band Jacobian. If F is linear in u_x and u_t, system is also linear with 3-band matrix eventhou is given generaly. Is there any solver eficient in solving linear equations with banded matrix given implicitly? (I hope Newton-Raphson is.) As initial guess for the solution we can use extrapolated values. If solving fails we can try value from the node on left or right (this could help on shocks).
With linear solver:
If F is linear, we expres (2.7) as

$$
\boldsymbol{A}\bar{u}^{n+1}_{:} = \bar{b}.
$$

$\boldsymbol{A}$ is $M \times M$ 3-diagonal. Functions for evaluation of $\boldsymbol{M}$ and $\bar{b}$ are generated during compilation. In runtime we solve just the linear system. In this aproach difference schema must be chosen before compilation of model.

Implicit solver and systems of PDE If we solve e.g. system with three variables u, v, w, se can sort difference equations in order

$$
u_1,\ v_1,\ w_1,\ u_2,\ v_2,\ w_2,\ u_3,\ v_3,\ w_3,\ ...
$$

so that the system is stil banded.

Chapter 3

Example models

3.1 Package PDEDomains

Modelica code of domain definitions:

```
package PDEDomains
  import C = Modelica.Constants;
  record DomainLineSegment1D
    parameter Real l = 1;
    parameter Real a = 0;
    function shapeFunc
      input Real v;
      output Real x = l*v + a;
    end shapeFunc;
    Region1D interior(shape = shapeFunc, range = {0,1});
    Region0D left(shape = shapeFunc, range = 0);
    Region0D right(shape = shapeFunc, range = 1);
  end DomainLineSegment1D;

  record DomainRectangle2D
    parameter Real Lx = 1;
    parameter Real Ly = 1;
    parameter Real ax = 0;
    parameter Real ay = 0;
    function shapeFunc
      input Real v1, v2;
      output Real x = ax + Lx * v1, y = ay + Ly * v2;
    end shapeFunc;
    Region2D interior(shape = shapeFunc, range =
       {{0,1},{0,1}});
    Region1D right(shape = shapeFunc, range = {1,{0,1}});
    Region1D bottom(shape = shapeFunc, range = {{0,1},0})
```

```
      ;
    Region1D left(shape = shapeFunc, range = {0,{0,1}});
    Region1D top(shape = shapeFunc, range = {{0,1},1});
end DomainRectangle2D;

record DomainCircular2D
    parameter Real radius = 1;
    parameter Real cx = 0;
    parameter Real cy = 0;
    function shapeFunc
      input Real r,v;
      output Real x,y;
    algorithm
      x:=cx + radius * r * cos(2 * C.pi * v);
      y:=cy + radius * r * sin(2 * C.pi * v);
    end shapeFunc;
    Region2D interior(shape = shapeFunc, range = {{O,1},{
      O,1}});
    Region1D boundary(shape = shapeFunc, range =
       {1,{0,1}});
end DomainCircular2D;

record DomainBlock3D
    parameter Real Lx = 1, Ly = 1, Lz = 1;
    parameter Real ax = 0, ay = 0, az = 0;
    function shapeFunc
      input Real vx, vy, vz;
      output Real x = ax + Lx * vx, y = ay + Ly * vy, z =
         az + Lz * vz;
    end shapeFunc;
    Region3D interior(shape = shapeFunc, range =
       {{0,1},{0,1},{0,1}});
    Region2D right(shape = shapeFunc, range =
       {1,{0,1},{0,1}});
    Region2D bottom(shape = shapeFunc, range =
       {{0,1},{0,1},1});
    Region2D left(shape = shapeFunc, range =
       {0,{0,1},{0,1}});
    Region2D top(shape = shapeFunc, range =
       {{0,1},{0,1},1});
    Region2D front(shape = shapeFunc, range =
       {{0,1},0,{0,1}});
    Region2D rear(shape = shapeFunc, range =
       {{0,1},1,{0,1}});
end DomainBlock3D;
//and others ...
```

```
end PDEDomains;
```

Listing 3.1: Standard domains deffinitions: 1D – Line segment, 2D – Rectangle, Circle, 3D – Block

3.2 Simple models

3.2.1 Advection equation (1D)[?]

L .. length
c .. constant, assume $c > 0$
$u \in \langle 0, L\rangle \times \langle 0, T\rangle \to \mathbb{R}$

equation

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0$$

initial conditions

$$u(x,0) = 1$$

boundary conditions

$$u(0,t) = \cos(2\pi t)$$

Modelica code

```
model advection "advection equation"
  import C = Modelica.Constants;
  parameter Real L = 1; // length
  parameter Real c = 1;
  parameter DomainLineSegment1D omega(length = L);
  field Real u(domain = omega, start = 1);
equation
  pder(u,time) + c*pder(u,x) = 0   in omega.
      interior;
  u = cos(2*C.pi*time)             in omega.left;
end advection;
```

Listing 3.2: Advection equation in Modelica

Flat model

```
/*TODO: finish it!!*/

model advection_flat "advection equation"
  import C = Modelica.Constants;
  parameter Real L = 1; // length
  parameter Real c = 1;
//  parameter DomainLineSegment1D omega(length = L
   );
  parameter Real DomainLineSegment1D.l = L;
  parameter Real DomainLineSegment1D.a = 0;
  function DomainLineSegment1D.shapeFunc
    input Real v;
    output Real x = l*v + a;
  end DomainLineSegment1D.shapeFunc;
  Domain1DInterior DomainLineSegment1D.interior(
     shape = shapeFunc, range = {0,1});
  Domain1DBoundary DomainLineSegment1D.left(shape
     = shapeFunc, range = {0,0});
  Domain1DBoundary DomainLineSegment1D.right(shape
     = shapeFunc, range = {1,1});

  field Real u(domain = omega, start = 1);
equation
  pder(u,time) + c*pder(u,x) = 0  in  omega.
     interior;
  u = cos(2*pi*time) in omega.left;
end advection_flat;
```

Listing 3.3: Advection equation – flat model

Generated C code

```
#include <math.h>
#include "model_data.h"
#include "PDESolver.h"
#include "model.h"

//#define _USE_MATH_DEFINES
//#include <math.h>
double pi = 3.14159265358979323846;
```

```
int setupArrayDimensions(struct MODEL_DATA* mData)
    {
    mData->nStateFields = 1;
    mData->nAlgebraicFields= 0;
    mData->nParameterFields= 0;
    mData->nParameters = 4;
    mData->nDomainSegments = 3;
    return 0;
}



int setupModelParameters(struct MODEL_DATA* mData)
{
    /*interior:*/
    mData->domainRange[0].v0 = 0;
    mData->domainRange[0].v1 = 1;
    /*left*/
    mData->domainRange[1].v0 = 0;
    mData->domainRange[1].v1 = 0;
    /*right*/
    mData->domainRange[2].v0 = 1;
    mData->domainRange[2].v1 = 1;
    /*advection.L*/mData->parameters[0] = 1;
    /*advection.c*/mData->parameters[1] = 1;
    /*DomainLineSegment1D.l*/mData->parameters[2]
       = 1;
    /*DomainLineSegment1D.a*/mData->parameters[3]
       = 0;
    mData->isBc[mData->nStateFields*0 + 0] = 1;
    mData->isBc[mData->nStateFields*0 + 1] = 0;
    return 0;
}

int setupInitialState(struct MODEL_DATA* mData){
    int i;
    for (i=0; i<mData->M; i++){
        //TODO: should be done generally, with
           some kind of stateInitial(x) function
           called from static code.(
        mData->stateFields[mData->M*0 + i] = 1;
    }
    return 0;
}

double shapeFunction(struct MODEL_DATA *mData,
    double v)
{
    return /*DomainLineSegment1D.l*/mData->
        parameters[2]*v + /*DomainLineSegment1D.a*/
        mData->parameters[3];
}


int functionPDE(struct MODEL_DATA *mData, int
    dScheme)
{
    int M = mData->M;
    int i;
    for (i = 0; i<M; i++)
        /*u_t*/mData->stateFieldsDerTime[M*0 + i]
            = - /*c*/mData->parameters[1] * /*u_x*/
            mData->stateFieldsDerSpace[mData->M*0 +
             i];
    return 0;
}

int functionBC(struct MODEL_DATA *mData)
{
    //should be writen generaly -- independent on
        particular grid
    int M = mData->M;
    mData->stateFields[M*0 + 0] = cos(2*pi*mData->
        time);
    return 0;
}

/*double eqSystemMaxEigenVal(struct MODEL_DATA*
    mData){
    return cmData->parameters[1];
}*/
```

Listing 3.4: Advection equation – "generated" C code

3.2.2 String equation (1D)[?]

L .. length

$u \in \langle 0, L\rangle \times \langle 0, T\rangle \to \mathbb{R}$ (string position)

c .. constant

equation:

$$\frac{\partial^2 u}{\partial t^2} - c\frac{\partial^2 u}{\partial x^2} = 0$$

initial conditions

$$\begin{aligned} u(x,0) &= \sin\left(\frac{4\pi}{L}x\right) \\ \dot{u}(x,0) &= 0 \end{aligned}$$

boundary conditions

$$u(0,t) = 0, \quad u(L,t) = 0$$

Modelica code

```
model string "model of a vibrating string with fixed ends
    "
  import C = Modelica.Constants;
  parameter Real L = 1; // length
  parameter Real c = 1; // tension/(linear density)
  parameter DomainLineSegment1D omega(length = L);
  function u0
    input Real x;
    output Real u0 := sin(4*C.pi/L*x);
  end u0;
  field Real u(domain = omega, start = u0);
initial
  pder(u,time) = 0;
equation
  pder(u,time,time) - c*pder(u,x,x) = 0    in omega.
      interior;
  u = 0;                                   in omega.left +
       omega.right;
end string;
```

Listing 3.5: String model in Modelica

Generated C code

```
#include <math.h>
#include "model_data.h"
#include "PDESolver.h"
#include "model.h"
#include "diff.h"

double pi = 3.14159265358979323846;

int setupArrayDimensions(struct MODEL_DATA* mData) {
    mData->nStateFields = 2;
    mData->nAlgebraicFields= 1;
    mData->nParameterFields= 0;
    mData->nParameters = 4;
    mData->nDomainSegments = 3;
    return 0;
}

int setupModelParameters(struct MODEL_DATA* mData)
{
    /*interior:*/
    mData->domainRange[0].v0 = 0;
    mData->domainRange[0].v1 = 1;
    /*left*/
    mData->domainRange[1].v0 = 0;
    mData->domainRange[1].v1 = 0;
    /*right*/
    mData->domainRange[2].v0 = 1;
    mData->domainRange[2].v1 = 1;
    /*string.L*/mData->parameters[0] = 1;
    /*string.c*/mData->parameters[1] = 1;
    /*DomainLineSegment1D.l*/mData->parameters[2] = 1;
    /*DomainLineSegment1D.a*/mData->parameters[3] = 0;
    mData->isBc[mData->nStateFields*0 + 0] = 1;
    mData->isBc[mData->nStateFields*0 + 1] = 1;
    return 0;
}

double /*u0*/function_0(struct MODEL_DATA* mData, double
   x){
    return sin(4*pi/ /*string.L*/mData->parameters[0]*x);
}

int setupInitialState(struct MODEL_DATA* mData){
    int i;
    for (i=0; i<mData->M; i++){
        /*u*/mData->stateFields[mData->M*0 + i] =
            function_0(mData, mData->spaceField[mData->M*0
             + i]);
        /*u_t*/mData->stateFields[mData->M*1 + i] = 0;
    }
    return 0;
}

double shapeFunction(struct MODEL_DATA *mData, double v)
{
    return /*DomainLineSegment1D.l*/mData->parameters[2]*
        v + /*DomainLineSegment1D.a*/mData->parameters[3];
}

// pder(u,t)    = u_t
// pder(u,x)    = u_x
// pder(u_t,t) = c pder(u_x,x)


int functionPDE(struct MODEL_DATA *mData, int dScheme)
{
    // both states and algebraics have their specific
        array for space derivatives

    // states u, u_t
    // algebraics u_x

    //we have u, u_t, pder(u,x), pder(u_t,x)

    // u_x          = pder(u,x)
    // pder(u_x,x)  = diff(u_x,x)
    // pder(u,t)    = u_t
    // pder(u_t,t)  = c pder(u_x,x)

    //u      stateFields[M*0
    //u_t    stateFields[M*1
    //u_x    algebraicFields[M*0


    int M = mData->M;
    int i;
    for (i = 0; i<M; i++){
        /*u_x*/mData->algebraicFields[M*0 + i] = mData->
            stateFieldsDerSpace[M*0 + i];
    }
    differentiateX(/*u_x*/&(mData->algebraicFields[M*0]),
        /*pder(u_x,x)*/&(mData->algebraicFieldsDerSpace[M
       *0]), mData, dScheme);
    for (i = 0; i<M; i++){
        /*pder(u,t)*/mData->stateFieldsDerTime[M*0 + i] =
            /*u_t*/mData->stateFields[M*1 + i];
        /*pder(u_t,t)*/mData->stateFieldsDerTime[M*1 + i]
              = /*c*/mData->parameters[1]* /*pder(u_x,x)*/
            mData->algebraicFieldsDerSpace[M*0 + i];
    }
  return 0;
  //in this approach some arrays for space derivatives
      might be unused (here pder(u_t,x))
}

//int functionPDE_2(struct MODEL_DATA *mData)
//{
//     // all space derivatives of states and algebraics
    are stored as different algebraic fields
//     //—————————————————————————————————————
//     // TODO: pokracovat
//     //—————————————————————————————————————
//
//     // states u, u_t
//     // algebraics u_x
//
//     //we have u, u_t, pder(u,x), pder(u_t,x)
//
//     // u_x            = pder(u,x)
//     // pder(u_x,x)    = diff(u_x,x)
//     // pder(u,t)      = u_t
//     // pder(u_t,t)    = c pder(u_x,x)
//
//     //u       stateFields[M*0
//     //u_t     stateFields[M*1
//     //u_x     algebraicFields[M*0
//
//
//     int M = mData->M;
//     int i;
//     for (i = 0; i<M; i++){
//          /*u_x*/mData->algebraicFields[M*0 + i] = mData
    ->stateFieldsDerSpace[M*0 + i];
//     }
//     diffx(/*u_x*/mData->algebraicFields[M*0], /*der(u_x
    ,x)*/mData->algebraicFieldsDerSpace[M*0]);
//     for (i = 0; i<M; i++){
//          /*pder(u,t)*/mData->stateFieldsDerTime[M*0 + i]
     = /*u_t*/mData->stateFields[M*1 + i];
//          /*pder(u_t,t)*/mData->stateFieldsDerTime[M*1 +
    i]  = /*c*/mData->parameters[1]* /*pder(u_x,x)*/mData
    ->algebraicFieldsDerSpace[M*0 + i];
//     }
//  return 0;
//  //this aproach is confusing as algebraics array is
    used for various fields
//}
```

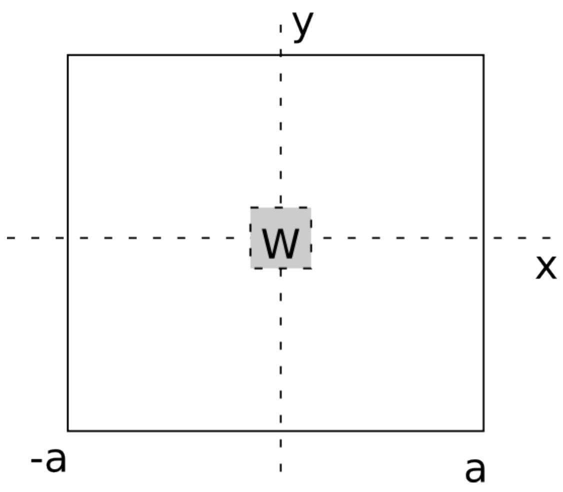

Figure 3.1: Heat eq.

```
int functionBC(struct MODEL_DATA *mData)
{
     int M = mData->M;
     mData->stateFields[mData->M*0 + 0] = 0;
     mData->stateFields[mData->M*0 + M-1] = 0;
     mData->stateFields[mData->M*1 + 0] = 0;
     mData->stateFields[mData->M*1 + M-1] = 0;

  return 0;
}
```

Listing 3.6: String equation – "generated" C code

3.2.3 Heat equation in square with sources (2D)

a .. domain square side hlaf length
c .. conductivity quocient
T .. temperature

$$W(x,y) \quad = \quad \begin{cases} 1 & \text{if } |x| < a/10 \text{ and } y < a/10 \\ 0 & \text{else} \end{cases}$$

equation

$$\frac{\partial T}{\partial t} - c\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) \quad = \quad W$$

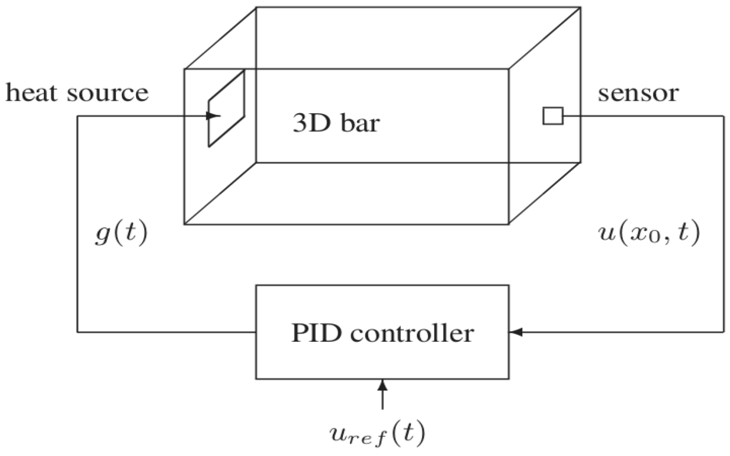

Figure 3.2: Heat transfer with source and PID controller

initial conditions

$$T(x, y, 0) = 0$$

boundary conditions insulated walls (top, left, bottom)

$$\begin{aligned}
\frac{\partial T}{\partial \bar{n}}(x, a, t) &= 0 \\
\frac{\partial T}{\partial \bar{n}}(-a, y, t) &= 0 \\
\frac{\partial T}{\partial \bar{n}}(x, -a, t) &= 0
\end{aligned}$$

fixed temperature (right)

$$T(a, y, t) = 0$$

3.2.4 3D heat transfer with source and PID controller [?]

new problems:

- system of ODE and PDE
- `in` operator used to acces field value in a concrete point (PID controler equation defining T_s).
- vector field
- differential operators `grad` and `diverg`

l_x, l_y, l_z .. room dimensions ($6m$, $4m$, $3.2m$)
T .. temperature (scalar field)
W .. thermal flux (vector field)
c .. specific heat capacity ($1012\,J \cdot kg^{-1} \cdot K^{-1}$
ϱ .. density of air ($1.2041\,kg \cdot m^{-3}$)
λ .. thermal conductivity ($0.0257\,W \cdot m^{-1}K$)
T_{out} .. outside temperature ($0\,°$C)
κ .. right wall heat transfer coefficient ($0.2\,W \cdot m^{-2} \cdot K^{-1}$
T_s .. temperature of the sensor placed in middle of the right wall
P .. power of heating
k_p, k_i, k_d .. coefficients of the PID controller (100, 200, 100)
T_d .. desired temperature (20°C)
e .. difference between temperature of the sensor and desired temperature

heat equation

$$\begin{aligned} \frac{1}{c\varrho}\nabla \cdot W &= -\frac{\partial T}{\partial t} \\ W &= -\lambda \nabla T \end{aligned}$$

boundary conditions left wall ($x = 0$) - heat flux given by heating power

$$W_x = \frac{P}{l_y l_z}$$

rare ($y = 0$) and front ($y = l_y$), resp. bottom ($y = 0$) and top ($z = l_z$) insulated walls

$$W_y = 0, \text{ resp. } W = 0$$

right wall ($x = l_x$) - not fully insulated

$$W_x = \kappa(T - T_{out})$$

PID controler

$$\begin{aligned} T_s &= T(l_x, \frac{l_y}{2}, \frac{l_z}{2}) \\ e &= T_d - T_s \\ P &= k_p e + k_i \int_0^t e(\tau)d\tau + k_d \frac{d}{dt} e \end{aligned}$$

Modelica code:

```
model heatPID
  record Room
    extends DomainBlock3D;
    Region0D sensorPosition(shape = shapeFunc, range =
        {{1,1},{0.5,0.5},{0.5,0.5}})   ;
  end Room

  parameter Real lx = 1, ly = 1, lz = 1;
  Room room(Lx=lx, Ly=ly, Lz=lz);
  field Real T(domain = room, start = Tout);
  field Real[3] W(domain = room, start = {0,0,0});
  parameter Real c = 1012;
  parameter Real rho = 1.204;
  parameter Real lambda = 0.0257;
  parameter Real Tout = 0;
  parameter Real kappa = 0.2;
  Real Ts;
  Real P;
  parameter Real kp = 100, ki = 200, kd = 100;
  parameter Real Td = 20;
  Real eInt;
 equation
  1/(c*rho)*diverg(W) = - pder(T,t)     in room;
  W = -lambda*grad(T)                    in room;
//TODO: use normal vector:
  W[1] = P/(lx*ly)                       in room.left;
  W[2] = 0                               in room.front +
      room.rare;
  W[3] = 0                               in room.top + room
      .bottom;
  W[1] = kappa*(T - Tout)                in room.right;
  Ts = T                                 in room.
      sensorPosition;
  e = Td - Ts;
  der(eInt) = e;
  P = kp*e + ki*eInt + kd*der(e);
end heatPid;
```

Listing 3.7: heat equation with PID controller

3.3 More complex realistic models

3.3.1 Henleho klička - protiproudová výměna

$c_{in}(x,t)$ .. koncentrace Na v sestupné části tubulu
$\bar{c}_{in}(x,t)$ .. koncentrace Na ve vzestupné části tubulu
$c_{out}(x,t)$ .. koncentraca Na v dření
$Q(x,t)$ .. tok vody v sestupné části tubulu
$f_{H_2O}(x,t)$ .. tok vody na milimetr délky z sestupné části tubulu do dřeně
f^*_{Na} .. tok sodíku ze vzestupné části tubulu do dřeně na milimetr délky – aktivní transport – parametr
L .. délka tubulu
P_{H_20} .. prostupnost cévy pro vodu (permeabilita)

$$\begin{aligned}
\frac{\partial Q}{\partial x}(x,t) + f_{H_20}(x,t) &= 0 \\
(c_{out}(x,t) - c_{in}(x,t)) \cdot P_{H_2O} &= f_{H_20}(x,t) \\
f_{H_20}(x,t) &= \frac{dV}{dt}(t) \\
Q(L,t) \cdot c_{in}(L,t) &= f^*_{Na} \cdot L + Q(L,t) \cdot c^*(t) \\
\frac{\partial}{\partial x}\left(\bar{c}_{in}(x,t) \cdot Q(x,t)\right) &= f^*_{Na} \\
f^*_{Na} \cdot L &= \frac{dm_{Na}}{dt}
\end{aligned}$$

3.3.2 Oxygen diffusion in tissue around vessel

polar coordinates (r,φ)

$$\begin{aligned}
\frac{\partial \varrho}{\partial t} + q\left(\frac{1}{r}\frac{\partial \varrho}{\partial r} + \frac{\partial^2 \varrho}{\partial r^2} + \frac{1}{r^2}\frac{\partial^2 \varrho}{\partial \varphi^2}\right) + w &= 0 \\
\varrho(r_0,\varphi) &= \varrho_0 \\
\varrho(r,0) &= \varrho(r,2\pi) \\
\varrho_{nnn}(R,\varphi) &= 0 \; (= \varrho_{tn}(R,\varphi))
\end{aligned}$$

ϱ .. oxygen concentration
ϱ_0 .. concentration in the vessel
q .. diffusion coefficient
w .. local oxygen consumption
R .. Ω diameter
The last equation should simulate infinite continuation of the domain.

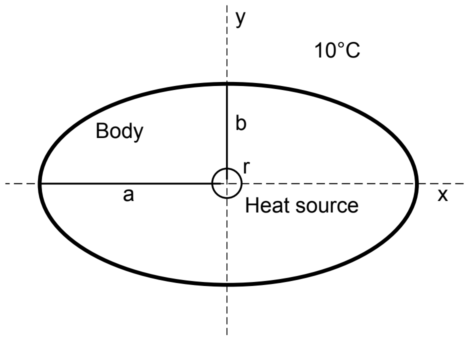

Figure 3.3: Scheme of heat diffusion in body

3.3.3 Heat diffusion

domain boundary

$$\partial\Omega \quad = \quad (a_b \cos(v),\, b_b \sin(v)),\ v \in \langle 0, 2\pi)$$

equation [?]

$$\frac{\partial T}{\partial t} + \frac{\lambda}{c\varrho}\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) \quad = \quad W$$

λ .. thermal conductivity
$W(x, y)$ .. heat power density of tissue (input)

$$W(x, y) = \begin{cases} W_0 & \text{if } x^2 + y^2 \leq r^2 \\ 0 & \text{else} \end{cases}$$

boundary condition

$$\lambda \frac{\partial T}{\partial n} = -\alpha(T - T_{out}),\ (x, y) \in \partial\Omega$$

α .. tissue-air thermal transfer coefficient [?]
initial condition

$$T(x, y, 0) = T_0(x, y)$$

3.3.4 Pulse waves in arteries caused by heart beats [?, ?]

$A(x, t)$ .. crossection of vessel
$U(x, t)$ .. average velocity of blood
$Q(x, t)$ .. flux

$$Q = AU$$

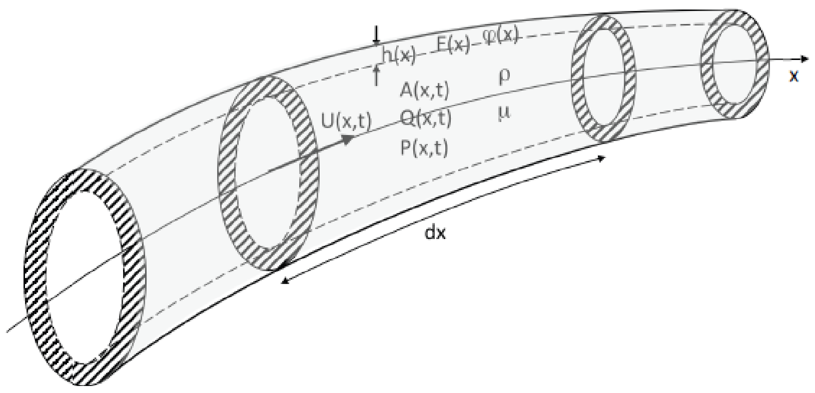

Figure 3.4: Arteria scheme

$P(x,t)$ .. pressure
P_{ext} .. external pressure
A_0 .. vessel crossection at $(P = P_{ext})$ (24mm)
$\beta = \frac{\sqrt{\pi} h_0 E}{(1-\nu^2) A_0}$
h_0 .. vessel wall thicknes (2mm)
E .. Young's modulus (0.24 - 6.55MPa)[**?**, **?**, **?**]
ν .. Poisson ratin (1/2)
$\varrho = 1050\,\mathrm{kg\,m^{-3}}$ .. blood density
$\mu = 4.0\,\mathrm{mPa\,s}$
α .. other ugly coefficient, let us say its 1
f .. frictional forces per unit length, let us assume inviscide flow $f = 0$

$$\begin{aligned}
\frac{\partial A}{\partial t} + \frac{\partial Q}{\partial x} &= 0 \\
\frac{\partial Q}{\partial t} + \frac{\partial}{\partial x}\left(\alpha \frac{Q^2}{A}\right) + \frac{A}{\varrho}\frac{\partial P}{\partial x} &= \\
= \frac{\partial Q}{\partial t} + \alpha\left(2\frac{Q}{A}\frac{\partial Q}{\partial x} - \frac{Q^2}{A^2}\frac{\partial A}{\partial x}\right) + \frac{A}{\varrho}\frac{\partial P}{\partial x} &= \frac{f}{\varrho} \\
P_{ext} + \beta\left(\sqrt{A} - \sqrt{A_0}\right) &= P
\end{aligned}$$

Three segment geometry – splitting arteria
We model arteria being splited into two minor arteries. Three same equation systems (super-indexes A, B, C) are solved on three different domains. Systems are connected via BC.

Boundary conditions

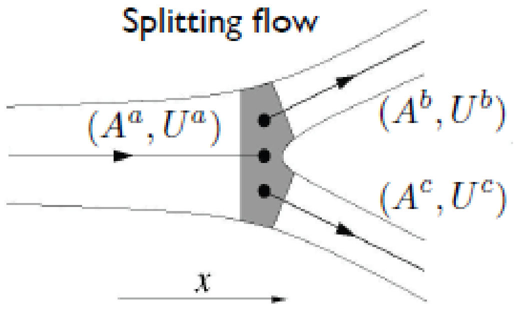

Figure 3.5: Arteria splitting

input

$$\begin{cases} P^A(0,t) = P_S & t \in \langle 0, \frac{1}{3}T_c) \\ Q^A(0,t) = 0 & t \in \langle \frac{1}{3}T_c, T_c) \end{cases}$$

T_c .. cardiac cycle period

junction

$$\begin{aligned} Q^A(L^A,t) &= Q^B(0,t) + Q^C(0,t) \\ P^A(L^A,t) &= P^B(0,t) \\ P^A(L^A,t) &= P^C(0,t) \end{aligned}$$

terminal we simulate the continuation of segments B and C with just a resitance

$$Q(L,t) = \frac{P(L,t)}{R_{out}}, \text{ for } B \text{ and } C$$

For check: the result should be in agreement with Moens–Korteweg equation.

Articles and books

I want to read:
A DIFFERENTIATION INDEX FOR PARTIAL DIFFERENTIAL-ALGEBRAIC EQUATIONS [?]
INDEX AND CHARACTERISTIC ANALYSIS OF LINEAR PDAE SYSTEMS [?]
Finite difference methods for ordinary and partial differential equations [?]
On the Role of Mathematical Abstractions for Scientific Computing[?]

Questions & problems:

Modelica language extension

- How to set initial condition for field derivative in similar way as using `start` atribute (i.e. not using equation in `initial` section)? See 3.2.2
- How to call divergence operator (standard `div` is is already used for integer division)
- **How to name space coordinates?** Should it be fixed or defined within the domain deffinition?
- Should it be possible to override initial and boundary conditions given in model with some different values from external configuration file?
- How should the shape, geometrical structure, mesh structure, etc. be described by an external file?

- Domain description where some parameters are in range and others are fixed: {{1,1}, {0.5,0.5}} or {{1,1}, 0.5}?
  - allow both
- How to deal with vector fields? How to acces its elements – using an index or scalar product with standard base vectors?
  - both
- How to distinguish the main domain (now called `DomainLineSegment1D`, `DomainRectangle2D` ...) and its "subsets" where some equations hold (now called `Domain0D`, `Domain1D` ...). I think only one of them should be called domain.
  - "subsets" renaimed to regions – `(Region0D, Region1D, Region 2D, Region3D)`

- Notation for normal vector to domain boundaries.
  - e.g. `omega.left.n`
- directional derivative
  - `der(u,v)` (u is scalar or vector field in R^n, v is vector in R^n)

Generated code

- How to represent on which particular boundary an boundary condition hold in generated code (or even on which interior domain hold which PDE equation system, if we support various interiors)? – Some domain struct could hold both shapeFunction parameter ranges and pointer (or some index) to function with the corresponding equations. Or boundary condition function knows on which elements (indexes) of variable arrays should be applied.
- **Should be generated functions independent on grid? It means either**
  ```
  functionPDEIndependent(u,u_x,t,x)
  u_t = ...
  return u_t
  ```
  or
  ```
  functionPDEDependent(data)
  for (int i ...)
  u_t[i] = ...
  ```

Numerics and solver

- **Shall we support higher derivatives in solver?**
- **What about space derivatives? – All state and algebraics have corresponding array for its space derivative, not all of them are used. – Or all space derivatives of states and algebraics are stored as different algebraic fields. – Or there is array for space derivatives that is utilised by both states and algebraics that need it.**
- What about multi step mothods (RK, P-K)?
- How to generate even (or arbitrary) meshes with nonlinea shape functions?
- How to generate mesh points just on the boundary? 1D – simple – just two points. 2D – We can go through the boundary curve and detect crossings of grid lines. 3D – who knows?!
- How to plugin an already existing solver?

- How to determine causality of boundary conditions and other equations that hold on less dimensional manifolds.
- Build whole solver in some PDE framework, perhaps Overture (http://www.overtureframework.org/)

TODO:

- Write a list of new concepts, key words etc in the language extension. How are they going to be translated and handled by the solver?
- Write a library for vector fields defining scalar and vector product, divergence, gradient, rotation...
- Write model in coordinates different from cartesian

Bibliography